19. First Person Camera

Here we will look at how to impliment a simple first person vector camera, to make it look like you are walking around. We will also learn how to go fullscreen, and exit without getting errors.

Introduction

In this lesson, we will be learning a way to implement a simple first person camera, which will enable us to look up, down, left, and right with the mouse, and move forward and back, and strafe left and right. It is not too difficult, so we will just get right to the code and explain whats going on as we go.
We will be using a seamless grass texture I made to texture the ground with grass, here it is:

Global Declarations

Here we have a bunch of new declarations. You can see four new vectors, two new matrices, and a couple floats. The first two vectors are describing the forward and right direction in the world. We will use these when calculating the rotation of our camera. The second two vectors are the forward and right directions of our camera. We will use the to move our camera around, so when we press the forward key, all we have to do is move along the camForward vector to make it look like our camera is walking straigt ahead.

The camRotationMatrix does not actually need to be created, since we could have just used the same rotation matrix we have been using for our cubes, but it is useful to rotate our camera. The second one is another matrix describing the world space of our "ground". Again, we could have used one of the cube world matrices, but this would be clearer.

The first two floats, moveLeftRight and moveBackForward, will be used to move along the camForward and camRight vectors when we want to move the camera forward or strafe right. The second two floats are used to calculate the rotation around the x and y axis for our camera when looking around.

```
XMVECTOR DefaultForward = XMVectorSet(0.0f,0.0f,1.0f, 0.0f);
XMVECTOR DefaultRight = XMVectorSet(1.0f,0.0f,0.0f, 0.0f);
XMVECTOR camForward = XMVectorSet(0.0f,0.0f,1.0f, 0.0f);
XMVECTOR camRight = XMVectorSet(1.0f,0.0f,0.0f, 0.0f);

XMMATRIX camRotationMatrix;
XMMATRIX groundWorld;

float moveLeftRight = 0.0f;
float moveBackForward = 0.0f;

float camYaw = 0.0f;
float camPitch = 0.0f;
```

New Function

This is a prototype of our new function, which will be used to update our camera.

```
void UpdateCamera();
```

Going Fullscreen

Here is our swapchain description. To go fullscreen, we have to set the Windowed member to false. We can go in and out of fullscreen by pressing "Alt+Enter".

```
swapChainDesc.BufferDesc = bufferDesc;
swapChainDesc.SampleDesc.Count = 1;
swapChainDesc.SampleDesc.Quality = 0;
swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
swapChainDesc.BufferCount = 1;
swapChainDesc.OutputWindow = hwnd;
///////////////**************new**************////////////////////
swapChainDesc.Windowed = false;
///////////////**************new**************////////////////////
swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_DISCARD;
```

The UpdateCamera() Function

Here is our UpdateCamera() function, which will be called every frame at the end of our DetectInput() function.

```
void UpdateCamera()
{
    camRotationMatrix = XMMatrixRotationRollPitchYaw(camPitch, camYaw, 0);
    camTarget = XMVector3TransformCoord(DefaultForward, camRotationMatrix );
    camTarget = XMVector3Normalize(camTarget);

    XMMATRIX RotateYTempMatrix;
    RotateYTempMatrix = XMMatrixRotationY(camYaw);

    camRight = XMVector3TransformCoord(DefaultRight, RotateYTempMatrix);
    camUp = XMVector3TransformCoord(camUp, RotateYTempMatrix);
    camForward = XMVector3TransformCoord(DefaultForward, RotateYTempMatrix);

    camPosition += moveLeftRight*camRight;
    camPosition += moveBackForward*camForward;

    moveLeftRight = 0.0f;
    moveBackForward = 0.0f;

    camTarget = camPosition + camTarget;

    camView = XMMatrixLookAtLH( camPosition, camTarget, camUp );
}
```

Rotating the Camera

The first three lines in our UpdateCamera() function will update the cameras target. The first line uses a new function from the xna math library, called XMMatrixRotationRollPitchYaw. This function has three parameters, the first is the pitch in radians to rotate, the second is the Yaw in radians to rotate, and the third is the Roll in radians to rotate. It returns a rotation matrix. This function is very usefull for cameras especially. you are able to rotate around all axis' at the same time, Yaw, Pitch, Roll. Yaw is the rotation left/right (y-axis), pitch is the rotation up/down (x-axis) and Roll is the rotation like your doing a cartwheel (z-axis). Our camera will be looking up/down and left/right, So we will be setting the yaw and pitch parameters of this function. Since we are not using the roll parameter, we set that to zero.

The next line sets our cameras Target vector by rotating the DefaultForward vector with the rotationMatrix we just created, and setting it to our Target matrix. After that, we normalize our vector, because the last operation may have made one or more of the Target vectors values greater than 1.0f, or less than -1.0f, in which case it would be not of unit length.

```
camRotationMatrix = XMMatrixRotationRollPitchYaw(camPitch, camYaw, 0);
camTarget = XMVector3TransformCoord(DefaultForward, camRotationMatrix );
camTarget = XMVector3Normalize(camTarget);
```

Now we need to find the new Right and Forward directions of our camera. We will do this using a rotation matrix which will be rotated on the Y-axis. Since our camera is a first person camera, and we only need to move in the X and Z directions, We need to keep our camera's forward and right vectors pointing only in the x and z axis. We will use those vectors to move our camera back, forward, left and right. First we create a new matrix, then we rotate that matrix using the yaw variable we have set when the mouse moves.

```
XMMATRIX RotateYTempMatrix;
RotateYTempMatrix = XMMatrixRotationY(camYaw);
```

Update the camForward, camUp, and camRight Vectors

Next we transform the camera's right, up, and forward vectors using the RotateYTempMatrix matrix we just defined, and rotate the default right, up, and default forward vectors then set the result in the right, up, and forward vectors.

```
camRight = XMVector3TransformCoord(DefaultRight, RotateYTempMatrix);
camUp = XMVector3TransformCoord(camUp, RotateYTempMatrix);
camForward = XMVector3TransformCoord(DefaultForward, RotateYTempMatrix);
```

Moving the Camera

Next we update the position of our camera using the two values, moveLeftRight, and moveBackForward, and the two vectors, Right, and Forward. To move the camera left and right, we multiply the moveLeftRight variable withthe Right vector, and add that to the Position. Then to move back and forward, we multiply moveBackForward with the Forward vector and add that to the Position too. The moveLeftRight and moveBackForward values will be calculated when direct input detects a certain key was pressed (in the case of this lesson A,S,D, or W).

After that we reset the moveLeftRight and moveBackForward variables.

```
camPosition += moveLeftRight*camRight;
camPosition += moveBackForward*camForward;

moveLeftRight = 0.0f;
moveBackForward = 0.0f;
```

Set the camView Matrix

Now we add the Position of our camera to the target vector, then update our View matrix by using the XMMatrixLookAtLH() D3D function. This function updates our View matrix we use to calculate our WVP matrix by using the cameras position, Target, and Up vectors.

```
    camTarget = camPosition + camTarget;

    camView = XMMatrixLookAtLH( camPosition, camTarget, camUp );
```

The DetectInput() Function

We have updated our DetectInput() function to update our camera when moving the mouse or pressing the W, S, A, or D keys. Notice the line before we start checking for input. We have a new variable called "speed". This is the speed our camera will move when we reposition it every frame. We take the time variable passed in when this function is called, and multiply it with our speed, so that our camera will move the exact same distance in one second whatever the frames per second are. Next we check if the W, S, A, and D keys were pressed, and update the moveLeftRight and moveBackForward accordingly. After that we check for the mouse input, where we will update the camYaw and camPitch depending on how much the mouse has moved in the x or y axis since the last frame. After all that, we update our camera by calling the UpdateCamera() function.

```
void DetectInput(double time)
{
    DIMOUSESTATE mouseCurrState;

    BYTE keyboardState[256];

    DIKeyboard->Acquire();
    DIMouse->Acquire();

    DIMouse->GetDeviceState(sizeof(DIMOUSESTATE), &mouseCurrState);

    DIKeyboard->GetDeviceState(sizeof(keyboardState),(LPVOID)&keyboardState);

    if(keyboardState[DIK_ESCAPE] & 0x80)
        PostMessage(hwnd, WM_DESTROY, 0, 0);

    float speed = 15.0f * time;

    if(keyboardState[DIK_A] & 0x80)
    {
        moveLeftRight -= speed;
    }
    if(keyboardState[DIK_D] & 0x80)
    {
        moveLeftRight += speed;
    }
    if(keyboardState[DIK_W] & 0x80)
    {
        moveBackForward += speed;
    }
    if(keyboardState[DIK_S] & 0x80)
    {
        moveBackForward -= speed;
```

Clean Up - Exiting Fullscreen

Since Direct3D can't exit properly from fullscreen directly, we need to do something extra before we actually exit. We need to take our application out of fullscreen and into windowed mode before we actually start cleaning up. We can do that in the CleanUp() function by calling the SetFullScreenState() method of our swapchain.

```
void CleanUp()
{
    ///////////////**************new**************////////////////////
    SwapChain->SetFullscreenState(false, NULL);
    PostMessage(hwnd, WM_DESTROY, 0, 0);
    ///////////////**************new**************////////////////////

    //Release the COM Objects we created
    SwapChain->Release();
    d3d11Device->Release();
    d3d11DevCon->Release();
    renderTargetView->Release();
    squareVertBuffer->Release();
    squareIndexBuffer->Release();
    VS->Release();
    PS->Release();
    VS_Buffer->Release();
    PS_Buffer->Release();
    vertLayout->Release();
    depthStencilView->Release();
    depthStencilBuffer->Release();
    cbPerObjectBuffer->Release();
    Transparency->Release();
    CCWcullMode->Release();
    CWcullMode->Release();

    d3d101Device->Release();
    keyedMutex11->Release();
    keyedMutex10->Release();
    D2DRenderTarget->Release();
    Brush->Release();
    BackBuffer11->Release();
    sharedTex11->Release();
    DWriteFactory->Release();
```

Change the Light Direction

Of course we don't have to do this, but to get the full color of our grass texture, we need to make sure the light is hitting it directly from above.

```
light.dir = XMFLOAT3(0.0f, 1.0f, 0.0f);
light.ambient = XMFLOAT4(0.2f, 0.2f, 0.2f, 1.0f);
light.diffuse = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);
```

Describing the Ground's Vertex and Index Buffers

Our ground wil be a simple square. Notice our texture coordinates though. This is to get a repeating texture (I created a seamless grass texture for this), instead of one very very very stretched out texture covering the whole ground.

```
    //Create the vertex buffer
    Vertex v[] =
    {
        // Bottom Face
        Vertex(-1.0f, -1.0f, -1.0f, 100.0f, 100.0f, 0.0f, 1.0f, 0.0f),
        Vertex( 1.0f, -1.0f, -1.0f,   0.0f, 100.0f, 0.0f, 1.0f, 0.0f),
        Vertex( 1.0f, -1.0f,  1.0f,   0.0f,   0.0f, 0.0f, 1.0f, 0.0f),
        Vertex(-1.0f, -1.0f,  1.0f, 100.0f,   0.0f, 0.0f, 1.0f, 0.0f),
    };

    DWORD indices[] = {
        0,  1,  2,
        0,  2,  3,
    };

    D3D11_BUFFER_DESC indexBufferDesc;
    ZeroMemory( &indexBufferDesc, sizeof(indexBufferDesc) );

    indexBufferDesc.Usage = D3D11_USAGE_DEFAULT;
    indexBufferDesc.ByteWidth = sizeof(DWORD) * 2 * 3;
    indexBufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;
    indexBufferDesc.CPUAccessFlags = 0;
    indexBufferDesc.MiscFlags = 0;

    D3D11_SUBRESOURCE_DATA iinitData;

    iinitData.pSysMem = indices;
    d3d11Device->CreateBuffer(&indexBufferDesc, &iinitData, &squareIndexBuffer);

    D3D11_BUFFER_DESC vertexBufferDesc;
    ZeroMemory( &vertexBufferDesc, sizeof(vertexBufferDesc) );

    vertexBufferDesc.Usage = D3D11_USAGE_DEFAULT;
```

Updated Texture

We are not using that brain texture now, we are going to use a nice seamless grass texture!

```
hr = D3DX11CreateShaderResourceViewFromFile( d3d11Device, L"grass.jpg",
    NULL, NULL, &CubesTexture, NULL );
```

Updating Our Scene

We are not using the cubes in this lesson, so we won't be updating their world spaces. Instead, we will scale our ground to be 1000 units wide and long, and move it down 10 units on the y axis.

```
void UpdateScene(double time)
{
    //Reset cube1World
    groundWorld = XMMatrixIdentity();

    //Define cube1's world space matrix
    ///////////////**************new**************////////////////////
    Scale = XMMatrixScaling( 500.0f, 10.0f, 500.0f );
    Translation = XMMatrixTranslation( 0.0f, 10.0f, 0.0f );

    //Set cube1's world space using the transformations
    groundWorld = Scale * Translation;
    ///////////////**************new**************////////////////////
}
```

Change the Background Color

I thought a pitch black background was somewhat boring, so i changed it to a dark shade of grey.

```
float bgColor[4] = { 0.1f, 0.1f, 0.1f, 1.0f };
```

Thats it for our simple camera! Hope you got some use out of this one!

Exercise:

1. Make a Free-Look Camera, where you are not restricted to moving around on the X and Z plane.